CSC 311 – Winter 2022-2023
Design and Analysis of Algorithms
9. Greedy Algorithms

Prof. Mohamed Menai
Department of Computer Science
King Saud University

Outline

- Greedy algorithm
- Counting money
- Fractional knapsack problem
- Minimum spanning tree – Prim - Kruskal
- Dijkstra’s shortest-path algorithm

2

Greedy Algorithm

Constructs a solution to an optimization problem piece by piece through a sequence of choices that are:

Defined by an objective function and a set of constraints

- feasible, i.e. satisfying the constraints
- locally optimal (with respect to some neig definition)
- greedy (in terms of some measure), and irrevocable

For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations.

- The problems that have a greedy solution are said to posses the greedy-choice property.

Optimization problems

- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

4

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for "normal" coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
- Approximations/heuristics:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

5

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make \$6.39, you can choose:
    - a \$5 bill
    - a \$1 bill, to make \$6
    - a 25¢ coin, to make \$6.25
    - a 10¢ coin, to make \$6.35
    - four 1¢ coins, to make \$6.39
- E.g., for US money, the greedy algorithm always gives the optimum solution

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

Fractional Knapsack Problem

- In fractional knapsack problem, where we are given a set S of n items, subject to, each item I has a *positive* benefit b_i and a *positive* weight w_i, and we wish to find the maximum-benefit subset that *doesn't exceed* a given weight W.
- We are also allowed to take *arbitrary fractions* of each item.

Fractional Knapsack Problem

- I.e., we can take an amount x_i of each item i such that $\begin{cases} 0 \le x_i \le w_i \text{ for each } i \in S \\ \sum_{i \in S} x_i \le W \end{cases}$

The *total benefit* of the items taken is determined by the *objective function*

$$\sum_{i \in S} b_i \left(\frac{x_i}{w_i} \right)$$

Fractional Knapsack Problem

Algorithm FractionalKnapsack(S, W):

Input: Set S of items, such that each item $i \in S$ has a positive benefit b_i and a positive weight w_i; positive maximum total weight W

Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i / w_i$ {***value index*** of item i}
$w \leftarrow 0$ {total weight}
while $w < W$ **do**
  remove from S an item i with highest value index {greedy choice}
  $a \leftarrow \min\{w_i, W - w\}$ {more than $W - w$ causes a weight overflow}
  $x_i \leftarrow a$
  $w \leftarrow w + a$

Fractional Knapsack Problem

- In the solution a heap-based *priority-queue* (PQ) is used to store the items of S, where the *key* of each item is its *value index*
- With PQ, each greedy choice, which removes an item with the greatest value index, takes $O(\log n)$ time
- The fractional knapsack algorithm can be implemented in time $O(n \log n)$.

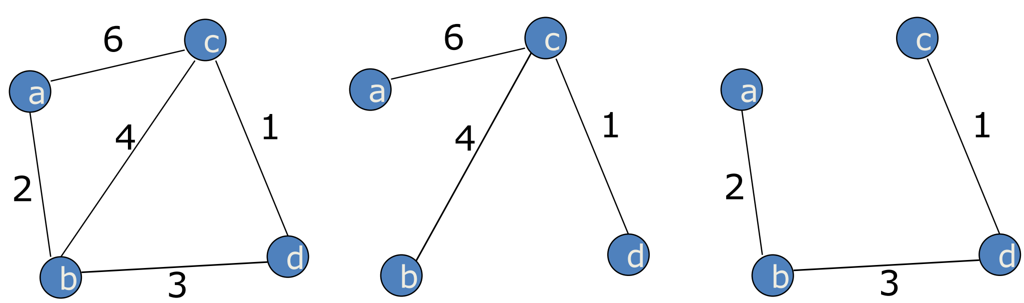
Fractional Knapsack Problem

- *Fractional knapsack problem* satisfies the *greedy-choice property*, hence
- Theorem: Given an instance of a fractional knapsack problem with set S of n items, we can construct a maximum benefit subset of S, allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time.

12

Minimum Spanning Tree (MST)

- *Spanning tree* of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices
- *Minimum spanning tree* of a weighted, connected graph *G*: a spanning tree of *G* of the minimum total weight



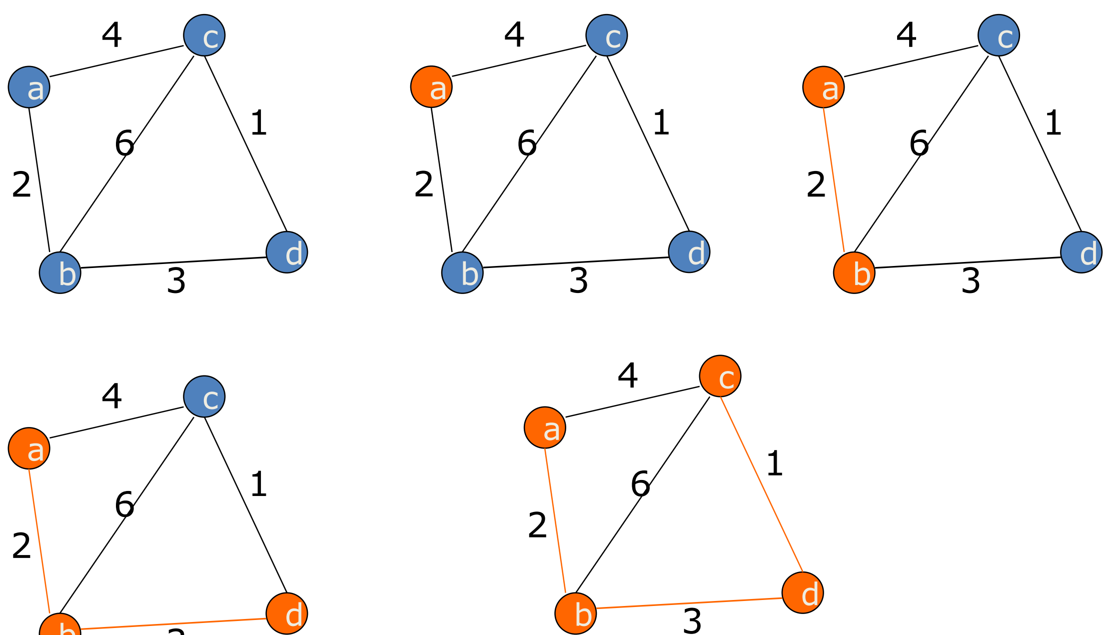
13

Prim's MST algorithm

- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees T_1, T_2, …, T_n
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)
- Stop when all vertices are included

14

Example



15

Notes about Prim's algorithm

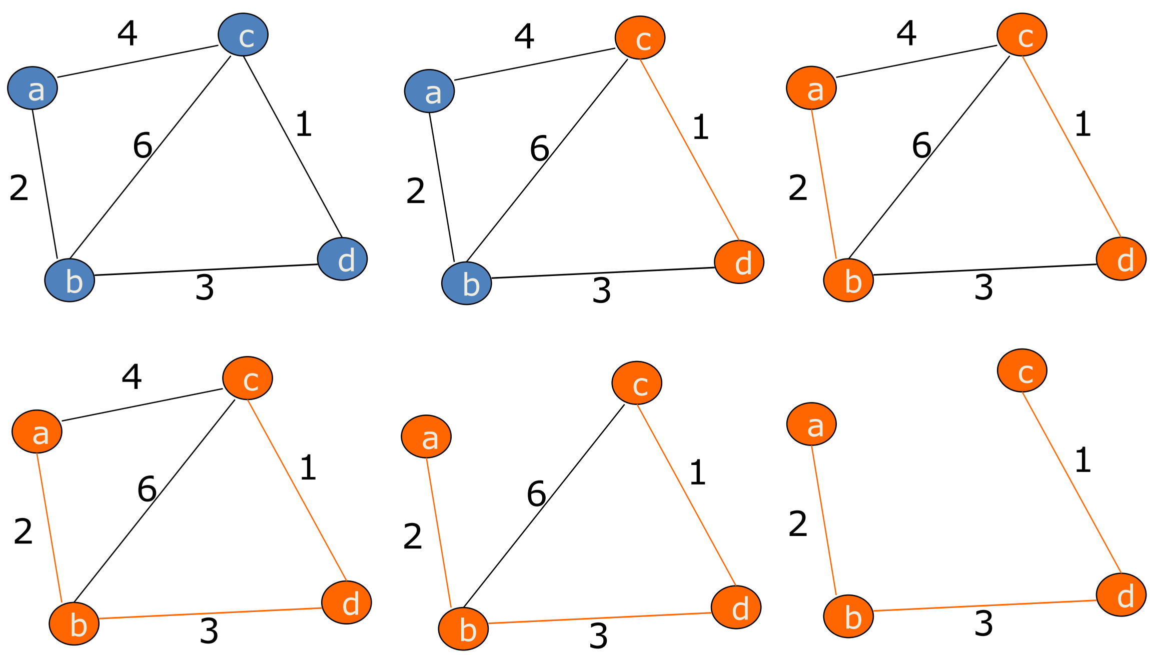
- Proof by induction that this construction actually yields an MST
- Needs priority queue for locating closest fringe vertex
- Efficiency
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$ for adjacency lists representation of graph with n vertices and m edges and min-heap implementation of the priority queue

16

Another greedy algorithm for MST: Kruskal's

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests F_1, F_2, …, F_{n-1}
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

Example



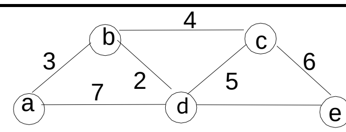
18

Notes about Kruskal's algorithm

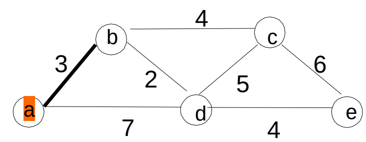
- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component
- Runs in $O(m \log m)$ time, with $m = |E|$. The time is mostly spent on sorting.

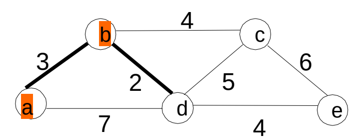
Dijkstra’s Shortest-Path Algorithm

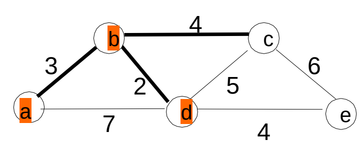
- Dijkstra’s algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  - Repeatedly (until all nodes are known),
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
      - If so, also reset the predecessor of the new node

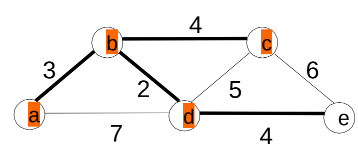


Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	









Notes on Dijkstra's algorithm

- Correctness can be proven by induction on the number of vertices.

We prove the invariants: (i) when a vertex is added to the tree, its correct distance is calculated and (ii) the distance is at least those of the previously added vertices.

- Doesn't work for graphs with negative weights (whereas Floyd's algorithm does, as long as there is no negative cycle).

- Applicable to both undirected and directed graphs

- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adjacent lists and min-heap implementation of priority queue

Reading

Chapter 9

Anany Levitin, Introduction to the design and analysis of algorithms, 3rd Edition, Pearson, 2011.